Breaking Database Lock-in: Agentic Regeneration of High Performance Storage Readers for Database Bypass

Victor Giannakouris
Cornell University
Ithaca, NY, USA
vg292@cornell.edu

Immanuel Trummer
Cornell University
Ithaca, NY, USA
itrummer@cornell.edu

ABSTRACT

Analytical workloads operating on data stored in external database systems face a fundamental bottleneck: data access is guarded entirely by the database driver, like JDBC or ODBC, forcing all reads through query execution and other driver layers that are not designed for bulk columnar analytics. Existing approaches, like JDBC and ODBC connectors, inherit this overhead, resulting in significant latency and throughput limitations for downstream external, analytic query engines. We present Jailbreak, an approach that bypasses the database engine entirely by reading storage files directly and materializing data as in-memory columnar buffers. Jailbreak's key insight is that database file formats, while complex, are fully specified by their source code and documentation, artifacts that Large Language Models (LLMs) can ingest to regenerate operator-specific table reading components without human-engineered parsing logic. Jailbreak leverages LLM-assisted code synthesis for database storage decoding, turning a traditionally opaque format into a directly queryable artifact. We evaluate Jailbreak on PostgreSQL and MySQL storage files, targeting analytical snapshot scenarios common in read replicas and offline processing pipelines. The generated reader produces Apache Arrow buffers consumable directly by most of the widely known query engines, including DuckDB, Apache Spark, and GPU-accelerated frameworks such as cuDF and Spark RAPIDS. We validate correctness against JDBC/ODBC-based baselines using the TPC-H benchmark across all query results, and demonstrate significant performance improvements in end-to-end analytical throughput, achieving up to 27× speedups. Our results suggest that LLM-assisted storage reader synthesis is a viable and generalizable methodology for breaking data lock-in across database systems, with applications beyond PostgreSQL and MySQL for any system whose file format is available to the LLM from documentation or source code.

VLDB Workshop Reference Format:
Victor Giannakouris and Immanuel Trummer. Breaking Database Lock-in: Agentic Regeneration of High Performance Storage Readers for Database Bypass. VLDB 2026 Workshop: Applied AI for Database Systems and Applications (AIDB 2026).

VLDB Workshop Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/gsvic/Jailbreak.

Proceedings of the VLDB Endowment. ISSN 2150-8097.

1 INTRODUCTION

The boundary between relational database systems (RDBMSes) and analytical query engines has long been guarded by the RDBMS engine itself. When an external system or application, like a GPU-accelerated framework, a vectorized query engine, or an ML feature store, needs data that resides in PostgreSQL or MySQL, it must issue SQL queries via a *wire protocol*, like JDBC or ODBC, wait for the server to traverse its storage layer, serialize rows through the wire protocol, and then deserialize those rows on the client side before any useful computation can begin. This architecture made sense when databases were monolithic systems that owned both storage and compute, but it is increasingly at odds with the modern data stack, in which a growing ecosystem of specialized engines, like DuckDB [23], DataFusion [4], cuDF [5], Spark SQL [27] or Velox [21], expects data in columnar, zero-copy formats such as Apache Arrow [2].

The overhead of this engine-mediated path is well documented. ConnectorX [24] showed that more than 85% of the wall-clock time in a typical `read_sql` call is consumed by client-side deserialization and format conversion, not by query execution or network transfer. Raasveldt and Mühleisen [22] demonstrated that existing wire protocols carry redundant metadata and impose expensive per-tuple serialization, proposing protocol-level redesigns to mitigate the cost. Federated query systems such as XDB [12], Wayang [7], Dingo [15], and MuSQLE [14] optimize *where* different parts of a query should be executed, yet they still rely on the wire protocol to external table transfer. In all of these approaches, the database server remains the sole gatekeeper of its own storage. No matter how aggressively the client or middleware is optimized, every byte of data must still pass through the wire protocol.

We observe that this bottleneck is not inherent to the data itself, but to the *access path*. Database storage formats, including PostgreSQL's heap files, MySQL's InnoDB `.ibd` pages, are complex, but they are not opaque. Their layouts are fully specified by a combination of source code, official documentation, and header files that have been stable across major releases. A reader that can parse these files directly and emit columnar in-memory byte streams would remove the server from the critical path entirely, turning what was an engine-mediated query into a simple file scan.

The main barrier to building custom table readers is engineering effort: hand-writing a correct parser for even one database's on-disk format requires deep knowledge of page layouts, which might include MVCC visibility rules, type encodings, and TOAST or overflow-page handling. Extending that parser to a second database doubles the effort, and keeping it current across versions adds a continuous maintenance burden. We argue that recent advances in Large Language Models (LLMs) change this calculus. An LLM can ingest the same documentation and source code that a human developer would consult and generate a format-specific reader in a single automated pass. If the generated code fails to compile or produces incorrect output, an agentic feedback loop can diagnose the error and regenerate the code, reducing what was a weeks-long systems programming task to a pipeline that completes in minutes.

In this paper we explore the ability of LLMs to synthesize custom, high-performance table readers for a target database system that read their files directly. First, we evaluate this idea by synthesizing two custom readers for Postgres and MySQL. We showcase that synthesizing custom table readers for existing database systems with LLMs is possible, and can achieve up to 27× speedups. Second, we present the vision of **Jailbreak**, an agentic approach that automates the synthesis of high-performance table readers that bypass the database engine entirely. Jailbreak-generated table readers ingest storage files directly and materialize table data as Apache Arrow columnar buffers. Jailbreak's multi-agent LLM pipeline takes a table schema and database type as input, and aims to produce a verified, plug-in-compatible shared library implementing a `pg_to_arrow` (or `mysql_to_arrow`) C interface. The pipeline comprises four specialized agents:

- A Dataset Generator that creates test tables consisting of various attributes and data types.
- An Architect that produces a structured table reader format specification.
- A Coder that generates high-performance, C++ reader code.
- A QA Tester that verifies correctness against known ground truth using the target database.

All the above are connected by an iterative feedback loop that refines the generated code until all checks pass or a retry budget is exhausted.

The resulting reader produces Arrow `RecordBatch` buffers that are consumable in a zero-copy fashion by the majority of analytical query engines, including DuckDB, Apache Spark, DataFusion, PyArrow, and GPU-accelerated frameworks such as cuDF and Spark RAPIDS. Because the reader operates on storage files rather than through the database server, it eliminates the JDBC/ODBC protocol and query execution overheads from the data path. We validate correctness against JDBC/ODBC-based baselines using the full TPC-H benchmark and demonstrate speedups of up to 5× for PostgreSQL and 27× for MySQL across six analytic query engines. To the best of our knowledge, Jailbreak is the first system to leverage LLM-assisted code synthesis for database bypass. Our experimental evaluation showcases that this methodology is viable and generalizable: any database system whose file format is recoverable from documentation or source code is a candidate for automated reader generation, opening a path toward breaking data lock-in without reverse engineering or vendor cooperation.

In summary, we make the following contributions:

- We identify direct storage-file access as a principled alternative to wire-protocol-based data export, removing the database engine from the analytical data path.
- We propose a four-agent LLM pipeline for automated synthesis of correct, type-complete C++ readers for PostgreSQL heap files and MySQL InnoDB pages, and demonstrate its viability through LLM-assisted interactive development.
- We produce Apache Arrow buffers that integrate zero-copy with six downstream engines, including GPU-accelerated frameworks.
- We evaluate two synthesized readers on TPC-H, demonstrating end-to-end ETL speedups up to 5.1× over PostgreSQL and up to 27× over MySQL wire-protocol baselines, with full correctness validation across all 22 TPC-H queries.

2 OVERVIEW

2.1 Jailbreak

Jailbreak is an agentic approach for synthesizing high-performance table readers for database system bypass. Jailbreak exploits the two following insights. First, the storage layer details of a database system can be ingested by an LLM both from the source code and documentation. For most of the popular open-source database systems, this information is already included in the already existing pretrained LLMs. Second, the LLM can utilize this knowledge in order to automatically synthesize high-performance code that can read the database system storage files directly, by completely bypassing the database system. This approach offers several performance benefits. First, it eliminates the wire protocol serialization/deserialization overheads, since the external application or query engine can read the storage files directly. Second, it leaves the target database system intact, since query execution that normally happens through JDBC/ODBC and adds extra load to the database system is completely eliminated. Third, it gives more control to the higher application layer on *how* to read and handle the data (e.g. partitioning schemes), since the synthesized reader works at the byte-level.

2.2 Implementation

We implement two shared libraries, one for Postgres (`pg_arrow.so`) and one for MySQL (`mysql_arrow.so`), each compiled from a single self-contained C++17 translation unit (approximately 800 lines each) with no external dependencies beyond the standard library. Both export a C-linkage function that reads a database storage file directly from disk and returns a fully typed Apache Arrow `RecordBatch` via the Arrow C Data Interface[1] without any involvement of the database server, wire protocol or serialization. In the following sections we present the abstract Arrow C Data Interface, as well as the implementation details of two synthesized table readers, including Postgres and MySQL.

2.3 Arrow C Data Interface

The Arrow C Data Interface defines two plain C structs, `ArrowSchema` and `ArrowArray`, that carry a complete columnar `RecordBatch` across language and library boundaries without copying the underlying buffers. Each library inlines the two struct definitions verbatim, making the compiled `.so` independent of any particular

[1] https://arrow.apache.org/docs/format/CDataInterface.html

Arrow SDK version. The consumer (Python, DuckDB, DataFusion, etc.) imports the batch with a single call:

```
batch = pa.RecordBatch._import_from_c(
            ctypes.addressof(array), ctypes.addressof(schema
     ))
```

This zero-copy handoff is the key performance property: the column buffers allocated by the C++ reader are owned by the `ArrowArray` through a release callback and are freed only when the consumer releases the batch.

Table 1 summarizes the type mapping used by both readers. Fixed-width types (booleans, integers, floats, dates, timestamps) use Arrow's two-buffer layout: a null bitmap and a packed data buffer. Variable-width types (strings, binary) use three buffers: a null bitmap, a 32-bit integer offset array, and a flat byte array for the character data.

Table 1: Source type to Arrow format string mapping.

Source type	**Arrow format**	**Description**
`bool`	`b`	1-bit boolean
`int2 / smallint`	`s`	int16
`int4 / int`	`i`	int32
`int8 / bigint`	`l`	int64
`float4 / real`	`f`	float32
`float8 / double`	`g`	float64
`date`	`tdD`	date32 (Unix epoch)
`timestamp`	`tsu:UTC`	timestamp[μs, UTC]
`numeric / decimal`	`g`	float64 (decoded)
`bpchar / varchar / text`	`u`	UTF-8 string

2.4 PostgreSQL Heap Reader

2.4.1 Page Layout. PostgreSQL stores each table as one or more *heap segment files* in `$PGDATA/base/dboid/reloid` (splitting at 1 GB boundaries with suffixes `.1`, `.2`, ...). Each file is a sequence of 8 KB pages. A page begins with a 24-byte `PageHeaderData` that contains `pd_lower`, the byte offset of the end of the line pointer array; `pd_upper`, the start of tuple data; and `pd_special`, the start of any special-purpose area (unused for heap pages). Following the header is an array of 4-byte *item identifiers* (`ItemId`), each encoding the page-relative offset, flags, and length of one tuple. Tuples are stored in reverse order from the top of the page downward.

2.4.2 MVCC Visibility. Each heap tuple begins with a 23-byte `HeapTupleHeader`. The implementation performs a lightweight MVCC visibility check before decoding: a tuple is skipped if its `t_xmax` (delete transaction ID) is non-zero *and* the `infomask` flags do not indicate that the deletion was rolled back (`HEAP_XMAX_INVALID`) or was only a row-lock rather than a real delete (`HEAP_XMAX_LOCK_ONLY`). This correctly excludes deleted and in-progress rows while retaining all committed live tuples, mirroring the visibility semantics of a serializable snapshot without requiring access to the transaction status tables.

2.4.3 Tuple Decoding. Each tuple's attribute data begins at byte offset `t_hoff`, which accounts for the header and any null bitmap. Attributes are decoded sequentially. Fixed-width types are aligned to their natural boundary (1, 2, 4, or 8 bytes) and copied directly from the page. Two epoch conversions are applied inline: PostgreSQL `date` stores days since 2000-01-01, so 10,957 days are added to obtain the Arrow `date32` (days since Unix epoch); similarly, `timestamp` values are shifted by $10{,}957 \times 86{,}400 \times 10^6$ microseconds.

Variable-length data (`bpchar`, `varchar`, `text`) uses PostgreSQL's *varlena* encoding. When the low-order bit of the first byte is set the datum is in "short" form: the remaining 7 bits of that byte encode the total datum size (including the 1-byte header), giving a maximum inline length of 127 bytes. Otherwise the full 4-byte header is read and the payload length is `(header » 2) - 4`. Trailing spaces are stripped from `bpchar` values before insertion into the Arrow character buffer.

`numeric` (arbitrary-precision decimal) is decoded to `float64`. PostgreSQL's internal numeric format stores digits in base 10,000 (*NBASE* groups), with a 2- or 4-byte header encoding the sign, the weight (most-significant NBASE group exponent), and the display scale; the number of digit groups is inferred from the remaining payload length. Both *short* (2-byte header, weight stored as a 6-bit magnitude with a sign-extension bit) and *long* (4-byte header, explicit 16-bit signed weight) formats are handled. The decode accumulates the digit groups and adjusts by $10{,}000^{\texttt{weight}-\texttt{ndigits}+1}$.

2.4.4 I/O Strategy. The reader buffers I/O in 2 MB batches (256 pages per `fread` call) to amortize system-call overhead. Column buffers are pre-allocated assuming approximately 6.5 million rows to avoid repeated reallocation during the scan. Row count is inferred from the first column's buffer length after the scan, so the caller does not need to provide it in advance.

2.4.5 BRIN-Pruned Scan. The `pg_to_arrow_brin` entry point accepts a path to the BRIN index file and a predicate of the form `"col:type:min:max"`, where either bound may be empty to indicate an unbounded range. The BRIN file is loaded entirely into memory (typically a few hundred kilobytes) and parsed as follows. Page 0 is the *meta page*: it holds the magic number (`0xA8109CFA`), the `pagesPerRange` value, and the index of the last revmap page. Pages 1 through `lastRevmapPage` are *revmap pages*, each storing an array of 6-byte `ItemPointerData` entries at a fixed layout; the capacity of a revmap page is $(\texttt{pd\_special} - 24)/6$ rather than being derived from `pd_lower`, which PostgreSQL does not update on revmap writes.

Each `ItemPointer` is followed to its `BrinTuple`. A critical subtlety in the `bt_info` field: the lower byte encodes two bits per stored attribute, bit $2i$ indicates that attribute i is entirely null (no valid min/max), while bit $2i + 1$ indicates the presence of *some* nulls but still valid extrema. A page range is pruned only when its summary tuple has valid (non-all-null) min and max values *and* the stored $[\texttt{brin\_min}, \texttt{brin\_max}]$ interval is disjoint from the predicate range. If either the min or max attribute is marked all-null, the BRIN summary is treated as uninformative and the range is conservatively scanned. Pruned page ranges are realized as `fseek` calls in `do_heap_scan`, providing genuine I/O reduction rather than merely parse-time filtering.

2.5 MySQL InnoDB Reader

2.5.1 Page Layout. MySQL InnoDB stores each table in a `.ibd` tablespace file composed of 16 KB pages. Each page begins with a 38-byte FIL header whose byte offsets 24–25 contain the 2-byte big-endian page type; only `FIL_PAGE_INDEX` pages (`0x45BF`) are processed. Index pages carry a 36-byte `PAGE_HEADER` followed by 20 bytes of file-segment (FSEG) headers, after which the infimum and supremum system records begin. The high bit of `PAGE_N_HEAP` at offset 42 indicates the *COMPACT* row format. `PAGE_LEVEL` at offset 64 distinguishes B-tree leaf pages (level 0) from interior nodes; only leaf pages contain row data.

2.5.2 Record Traversal. Within a leaf page, records form a singly linked list ordered by primary key. The *infimum pseudo-record* at page offset 99 serves as the list head; its 5-byte record header stores a big-endian signed 16-bit *next record* offset at bytes 3–4. Adding this offset to the current record's *origin pointer* (the address immediately following the header) gives the next record's origin. Traversal terminates at the *supremum* pseudo-record at offset 112 or when the next-record field is zero. Records flagged deleted (bit 5 of the header's info_bits byte) and records whose type field (the low 3 bits of the third header byte) is non-zero — that is, any record other than an ordinary leaf row, such as B-tree node pointers or the system pseudo-records — are skipped without decoding.

2.5.3 Compact Row Format Prefix. Before each record's origin, InnoDB stores two structures in reverse column order that are essential for correct decoding. First, the *variable-length prefix array*: one or two bytes per variable-length column (one byte if the maximum byte length $\leq$ 255, two bytes otherwise), stored so the entry for the *first* variable-length column is closest to the origin. Second, the *null bitmap*: one bit per nullable column, also in reverse order, stored immediately before the 5-byte record header. Because the first variable-length column lies nearest the origin, the implementation reads these lengths by starting just before the null bitmap (adjacent to the record header) and decrementing its cursor backward through memory for each successive column, rather than scanning forward.

2.5.4 Type Decoding. InnoDB stores all numeric types in *big-endian* byte order with a sign flip applied to enable byte-wise comparison. For signed integers the sign bit is XOR'd with 1 (e.g., `0x80000000` for `int4`), so that -2^{31} maps to `0x00000000` and $+2^{31}-1$ maps to `0xFFFFFFFF`. Floating-point values use a two-case XOR: if the sign bit is set (the value is positive in IEEE 754), only the sign bit is flipped; otherwise all bits are inverted. After de-mangling the bytes are reinterpreted natively via `memcpy`.

`DECIMAL(M,D)` is stored in MySQL's binary decimal format. The integer and fractional parts are each split into groups of nine decimal digits, where each full group occupies 4 bytes and a partial group occupies 1–4 bytes according to the lookup table `DIG2BYTES`$[0..9] = [0, 1, 1, 2, 2, 3, 3, 4, 4, 4]$. For the integer part the partial group is the most-significant one and is stored first; for the fractional part the partial group is the least-significant one and is stored last. The sign is encoded in the high bit of the first byte (1 = positive) and removed before decoding; negative values additionally require flipping all bytes.

`DATE` occupies 3 bytes with the sign encoded in the high bit of the first byte (XOR `0x80`). The remaining 23 bits pack year (14 bits), month (4 bits), and day (5 bits). These are extracted and converted to a Julian Day Number, then adjusted to the Unix epoch by subtracting the JDN of 1970-01-01 (2,440,588), yielding an Arrow `date32` value.

InnoDB's *hidden columns* — the 6-byte transaction ID (`TRX_ID`) and 7-byte rollback pointer (`ROLL_PTR`) that separate primary-key data from non-key data in the physical record — are handled via a `raw:13` skip entry in the column specification, which advances the data pointer without emitting any output column.

2.6 Shared Design Principles

Both readers are intentionally structurally similar. Column specifications are passed as a comma-separated string of `"name:type"` pairs that is parsed at runtime, making the library table-agnostic. All heap or tablespace traversal, type decoding, and Arrow buffer construction happens in a single sequential pass per file, minimizing memory traffic. Buffers for fixed-width and variable-width columns are maintained as `std::vector` containers and transferred to the Arrow structs by pointer without copying. The release callback registered on the top-level `ArrowArray` recursively frees all child `ArrowArray`/`ArrowSchema` objects and the owning `BatchData` allocation, so the caller is responsible only for releasing the root struct.

3 AGENTIC C++ READER GENERATION

In order to automate the process of synthesizing a table reader, we present a multi-agent pipeline that takes a table name as input and outputs a verified, plug-in-compatible shared library (`pg_arrow_agent.so`) that implements the same `pg_to_arrow` C interface as the hand-written reader described in Section 2.

3.1 Pipeline Overview

The pipeline consists of four specialized agents that execute in sequence, with an iterative feedback loop between the code-generation and verification stages (Figure 1).

Dataset Generator. Rather than developing against a full benchmark table, the pipeline first creates a small synthetic test table ($\approx$50 rows) with the same schema as the target table of the database. However, the resulting reader can be used and it's compatible with any table in the database. Row values are generated deterministically, so the exact expected aggregates (min, max, sum for numeric columns; distinct count for string columns; min/max for dates) are known *before* the reader runs. This makes correctness verification a simple arithmetic comparison rather than a probabilistic check.

Architect. Given the database type and table schema (retrieved via `information_schema`), the Architect agent produces a structured JSON *format spec* that encodes all information the Coder agent needs: page-level constants (page size, header offsets, magic numbers), per-column type encodings (endianness, decode formulae, varlena header layout), Arrow format strings for each column, MVCC visibility rules, and a curated list of known non-obvious bugs (`known_gotchas`). The agent's system prompt is grounded with a project knowledge base (`knowledge/pg_format.md`) covering the

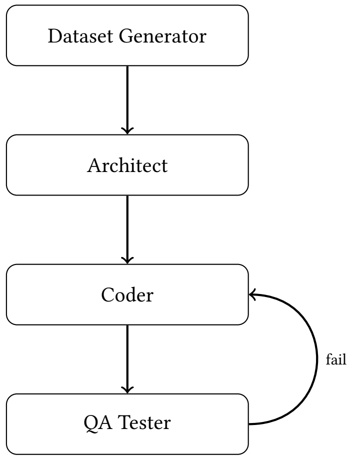

Figure 1: Agentic pipeline. The Coder–QA Tester loop repeats until the reader passes all correctness checks or the retry budget is exhausted.

PostgreSQL heap page layout and the Arrow C Data Interface buffer requirements.

Coder. The Coder agent receives the format spec and generates a complete, self-contained C++ source file. The output must compile with `g++ -O3 -std=c++17 -shared -fPIC` and export the following C function:

```
int pg_to_arrow(const char* heap_path, const char* col_spec,
                ArrowSchema* out_schema, ArrowArray*
     out_array,
                char* errmsg, int errmsg_len);
```

The agent is required to (i) parse `col_spec` at runtime rather than hardcoding column names or types, and (ii) implement all twelve standard PostgreSQL types (`bool`, `int2/4/8`, `float4/8`, `numeric`, `date`, `timestamp`, `bpchar`, `varchar`, `text`) regardless of which subset appears in the target table. The system prompt explicitly forbids prose output; the first character of the response must be `#` or `/` (a preprocessor directive or comment), and a post-processing step strips any residual natural-language text before the source is written to disk.

On retry attempts the agent additionally receives the full compiler output, the runtime output, and the complete history of prior QA diagnoses. If the same error type recurs across two or more consecutive attempts, the prompt instructs the agent to discard the current subsystem and rewrite it from scratch using a different strategy. A hash of each generated source is tracked to detect stuck loops in which the model produces identical code despite receiving failure feedback; in such cases an explicit rewrite hint is injected into the next prompt.

QA Tester. The QA Tester agent compiles the generated source, loads the resulting shared library via `ctypes`, runs it against the synthetic heap file, and imports the Arrow output as a PyArrow `RecordBatch`. Correctness is checked deterministically: row count and per-column statistics are compared against the Python-computed ground truth with a relative tolerance of 10^{-3} for floating-point values. Only if a mismatch is detected does the agent invoke an LLM to interpret the failure; the LLM is shown the raw compiler output, the runtime output, the ground truth, and the history of already-fixed bugs, and is asked to produce a structured JSON diagnosis with categorized new bugs and confirmed fixes. This design keeps LLM calls on the critical path to a minimum: a correctly functioning reader requires no LLM invocation in the QA step at all.

3.2 Retry Loop and Checkpointing

The Coder and QA Tester agents alternate for up to $N_{\max}$ attempts (default $N_{\max} = 10$). After every failed attempt the full pipeline state, including format spec, heap path, column specification, ground truth, complete feedback history, and the last generated source, is serialized to `outputs/table_checkpoint.json`. If the retry budget is exhausted without a passing reader, a subsequent invocation with `–resume` reloads this checkpoint, skips the Dataset Generator and Architect stages, and starts a fresh $N_{\max}$-attempt loop with the accumulated feedback history intact.

3.3 Compatibility and Deployment

The generated library is designed to be a drop-in replacement for the hand-written reader. It exports the identical `pg_to_arrow` symbol with the same function signature, the same inline `ArrowSchema` and `ArrowArray` struct layout, and the same zero-copy Arrow C Data Interface contract. Upon a successful run the pipeline automatically copies the winning shared library, e.g. `pg_arrow_agent.so`, in the project root, making it immediately usable by all existing benchmarks.

3.4 Multi-Provider Support

All agents are invoked through LiteLLM, which provides a unified interface over OpenAI, Anthropic, AWS Bedrock, and local Ollama models. Each of the three LLM-backed agents (Architect, Coder, QA Tester) can be assigned an independent model via command-line flags, enabling heterogeneous configurations such as using a reasoning-optimized model for code generation while routing cheaper calls to a smaller model for diagnosis. All agents run at temperature 0 to maximize determinism.

4 EXPERIMENTAL EVALUATION

We present an early experimental evaluation of Jailbreak, in which we benchmark an end-to-end pipeline that extracts a table from a database system and exports it as a local Parquet file. We use Postgres and MySQL as the target database system and multiple query engines, including DuckDB, DataFusion, PyArrow, Spark SQL, as well as GPU-accelerated query engines including cuDF and Spark RAPIDS.

4.1 Setup

All of our experiments were run on a g4dn.xlarge AWS instance with 4 vCPUs, 16 GiB of memory, 1 NVIDIA T4 GPU, and 125 GB of local NVMe SSD storage. We used TPC-H 1GB as our main benchmark, and we ran an end-to-end extraction of the *lineitem* table from the target database to a local Parquet file. All LLM agents were invoked via LiteLLM. Experiments used Claude Sonnet 4.6 (Anthropic).

4.2 ETL Benchmark: Database to Parquet

We evaluate end-to-end ETL performance by reading the TPC-H `lineitem` table (6,001,215 rows, ≈880 MB on disk) and writing the result to a Parquet file. Each method is run three times in an isolated subprocess; we report the best wall-clock time. We compare two data-access strategies across six analytics engines:

- **Jailbreak** — the heap file is read directly via `pg_to_arrow` / `mysql_to_arrow`, producing an Apache Arrow `RecordBatch` that is handed off zero-copy to the downstream engine.
- **Wire Protocol** — the engine reads from the database through the standard client protocol (PostgreSQL wire protocol or MySQL JDBC/connector).

4.3 PostgreSQL

Figure 2 shows results for PostgreSQL. Jailbreak consistently outperforms the wire protocol across all engines. DuckDB is the fastest overall at **5.46 s** via Jailbreak, compared to 9.58 s over wire (1.76× speedup). The largest gains appear with PyArrow and DataFusion, where the wire protocol incurs 23–24 s due to server-side serialization, while Jailbreak reads the same data in under 7 s (3.5× speedup). cuDF achieves the best Jailbreak time at **3.98 s**, benefiting from GPU-accelerated Parquet encoding after an Arrow handoff. Spark and Spark RAPIDS are the slowest engines in both modes (32–55 s), reflecting JVM startup and shuffle overhead rather than I/O cost.

4.4 MySQL

Figure 3 shows results for MySQL. The Jailbreak times are broadly similar to PostgreSQL (5.3–38.7 s), confirming that the bottleneck is the downstream engine rather than the source database. The wire protocol results, however, diverge dramatically: PyArrow via the MySQL Python connector takes **228.6 s**, 27× slower than the 8.5 s Jailbreak path, and this dominates the chart visually. DuckDB wire takes 44.2 s vs. 6.8 s (6.5×). The disparity is larger than in PostgreSQL because the row-based MySQL connector serializes each row individually, whereas the C++ reader decodes raw InnoDB pages at memory bandwidth. Only DataFusion wire (19.5 s) and cuDF wire (16.7 s) are closer to their C++ counterparts, as both use the ConnectorX driver which batches rows more aggressively.

4.5 Speedup Summary

Figure 4 summarizes the speedup of Jailbreak over the wire protocol for both databases. For PostgreSQL the speedup ranges from 1.3× (Spark RAPIDS) to 5.1× (cuDF), reflecting the relatively efficient PostgreSQL wire protocol. For MySQL the gains are larger across the board, peaking at **27×** for PyArrow and 6.5× for DuckDB. The results demonstrate that bypassing the database server is most impactful when the wire protocol is the primary bottleneck, as is the case for row-oriented connectors such as MySQL JDBC, and remains beneficial even for well-optimized protocols such as PostgreSQL's binary COPY format.

5 RELATED WORK

Jailbreak relates to four lines of prior work: efficient data transfer out of database systems, raw and in-situ query processing, automatic generation of systems code, and cross-system (polystore) data migration.

Fast data transfer and modern access paths. The inefficiency of engine-mediated data access is well established. Raasveldt and Mühleisen [22] showed that existing wire protocols impose redundant metadata and expensive per-tuple serialization, arguing for protocol-level redesign, while ConnectorX [24] identified client-side deserialization as the dominant cost of database-to-dataframe transfer with over 85% of wall-clock time in a typical `read_sql` call, and mitigated it with parallel partitioned reads. A complementary line of work builds dedicated cross-system transfer engines: XDBC [11, 13] decouples and pipelines the stages of data movement to saturate available hardware, and learned tuners automatically configure transfer parameters across heterogeneous environments [9]. On the access-path side, columnar protocols are displacing row-oriented JDBC/ODBC: Apache Arrow Flight SQL [3] returns query results directly as Arrow `RecordBatches` over an Arrow-native RPC layer, and Arrow Database Connectivity (ADBC) [1] exposes a JDBC/ODBC-style API whose native result type is already columnar Arrow, eliminating the row-to-column transposition those legacy drivers impose. All of these still route every byte through the database server; Jailbreak instead removes the server from the data path entirely and emits the same Arrow batches Flight SQL and ADBC deliver, making it complementary on the consumption side.

Raw and in-situ data processing. In-situ processing avoids the up-front cost of loading by querying files in their original format. NoDB and its PostgresRaw prototype [6] query raw text files directly, amortizing parsing through positional maps and caching; adaptive RAW processing [18] extends this to heterogeneous formats with just-in-time access paths, and SCANRAW [8] overlaps raw-file scanning with speculative, gradual loading in a parallel pipeline. Karpathiotakis et al. [19] further study raw access across a complex data ecosystem. Jailbreak shares the in-situ philosophy of reading data where it lives, but its input is the binary on-disk format of a live transactional database, like PostgreSQL heap and InnoDB pages carrying MVCC visibility metadata, rather than self-describing text, and its output is a zero-copy Arrow batch.

Synthesized systems. Rather than hand-writing an adapter for each system pair, PipeGen [16] automatically generates efficient binary data pipes by extending each DBMS's existing CSV/JSON import and export paths. Jailbreak generalizes this idea from transfer code to storage decoding: instead of reusing a system's own export routines, it synthesizes a reader for the raw on-disk format directly, driven by an LLM agentic pipeline that grounds generation in source code and documentation and verifies each candidate against a deterministic oracle (Section 3). Concurrent work applies LLM synthesis elsewhere in the stack: GenDB [20] envisions synthesizing entire query pipelines, and Wehrstein et al. [25] synthesize workload-specific OLAP engines. These target query execution; to the best of our knowledge, Jailbreak is the first to apply LLM code synthesis to database storage decoding.

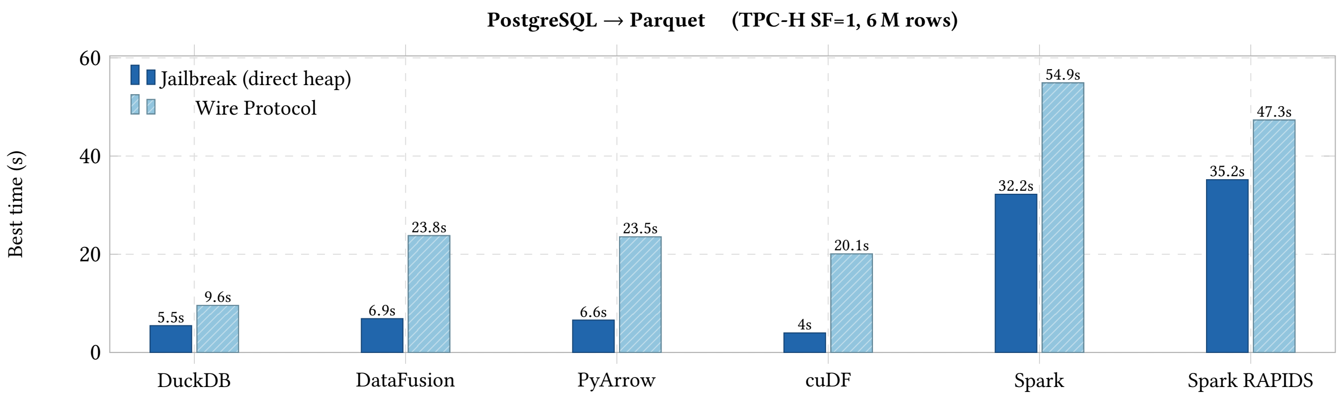

Figure 2: PostgreSQL ETL benchmark: time to read `lineitem` and write Parquet using Jailbreak (direct heap access) vs. wire protocol, per analytics engine. Lower is better. Best of 3 runs.

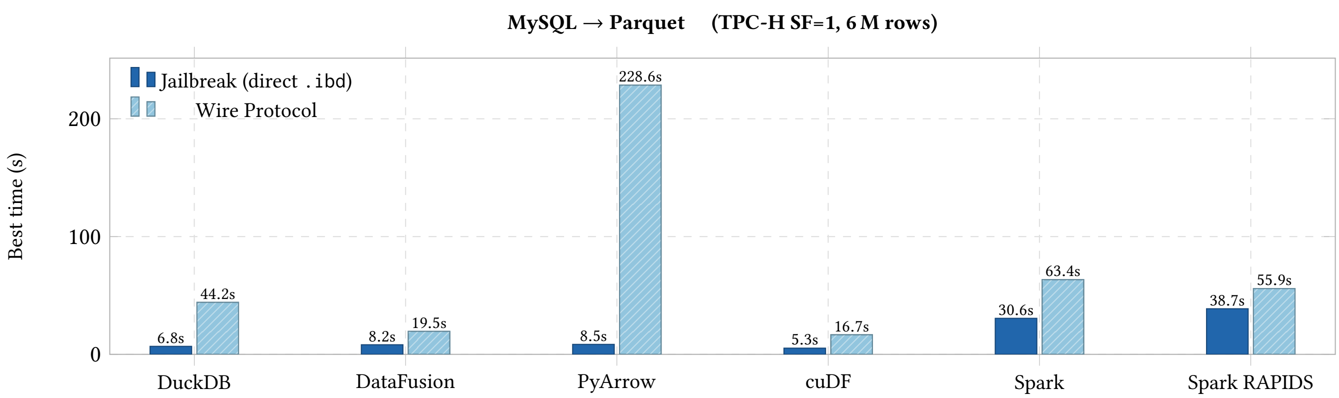

Figure 3: MySQL ETL benchmark: direct `.ibd` access vs. wire protocol per engine. PyArrow wire protocol takes 228 s vs. 8.5 s via the Jailbreak (27× speedup), dominating the y-axis. Best of 3 runs.

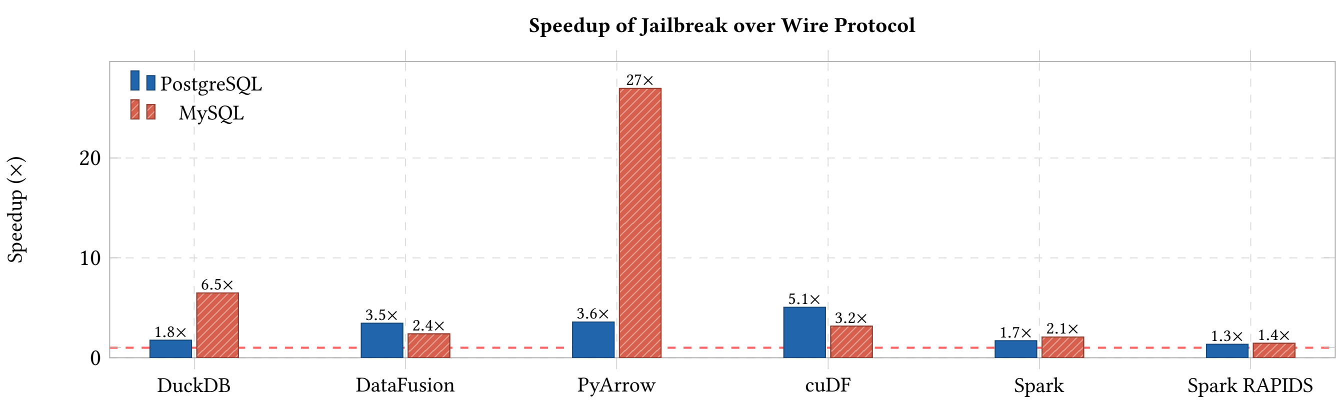

Figure 4: Speedup of Jailbreak direct heap access over wire protocol per engine. MySQL PyArrow achieves 27× because the MySQL wire protocol (JDBC/connector) is substantially slower than PostgreSQL's.

Polystore migration and federated query processing. Federated and polystore systems such as XDB [12], Wayang [7], MuSQLE [14], and pluggable federated optimizers [15] push query fragments to heterogeneous engines, optimizing *where* computation runs while still moving data over wire protocols. A related body of work targets the migration cost itself: Dziedzic et al. [10] characterize data transformation and migration across stores, Muses [17] provides a distributed migration engine for polystores, and FastDAWG [26] accelerates cross-engine migration in the BigDAWG polystore. These systems extract data through each engine's client interface; Jailbreak is orthogonal and can serve as a fast extraction primitive beneath them, turning a database's files into Arrow batches without engine cooperation and thereby replacing the JDBC/ODBC hop in a migration path.

Analytical engines and columnar formats. The emergence of embeddable analytical engines such as DuckDB [23], DataFusion [4], Velox [21], Spark SQL [27], and GPU-accelerated frameworks such as cuDF [5] has raised expectations for zero-copy, columnar data delivery, with Apache Arrow [2] as the de facto in-memory interchange format. Jailbreak is designed around this standard: its generated readers emit Arrow `RecordBatch`es via the Arrow C Data Interface, making them immediately consumable by any Arrow-native engine without format conversion.

6 CONCLUSIONS

We have presented **Jailbreak**, a system and methodology for breaking database lock-in by synthesizing high-performance storage readers that bypass the wire protocol entirely. Our key insight is that database file formats, while complex, are fully recoverable from source code and documentation, artifacts that LLMs can ingest to generate format-specific readers without human-authored parsing logic.

We evaluated two LLM-assisted readers for PostgreSQL and MySQL, demonstrating end-to-end ETL speedups of up to 5.1× and 27× respectively over wire-protocol baselines across six analytical engines, with full correctness validation on all 22 TPC-H queries. These results suggest that LLM-assisted storage layer synthesis is a viable and generalizable methodology: any database system whose file format is recoverable from documentation or source code is a candidate for automated reader generation.

Beyond the readers evaluated here, we have implemented the four-agent pipeline described in Section 3 and are actively working toward a fully automated, push-button workflow. Our immediate next steps include a systematic evaluation of the pipeline's convergence behavior, characterizing retry distributions, success rates, and generation latency across a broader set of schemas and database versions, as well as an ablation study assessing the contribution of each agent to final reader correctness. We are also exploring extension to additional database systems, including SQLite and MongoDB, where file format specifications are similarly recoverable from documentation and source code. We view Jailbreak as a first step toward a broader vision in which data lock-in across heterogeneous database systems can be broken without reverse engineering or vendor cooperation.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Award No. 2239326.

REFERENCES

[1] [n.d.]. ADBC: Arrow Database Connectivity. https://arrow.apache.org/adbc/.
[2] [n.d.]. Apache Arrow. https://github.com/apache/arrow.
[3] [n.d.]. Apache Arrow Flight SQL. https://arrow.apache.org/docs/format/FlightSql.html.
[4] [n.d.]. Apache DataFusion. https://github.com/apache/datafusion.
[5] [n.d.]. cuDF. https://github.com/rapidsai/cudf.
[6] Ioannis Alagiannis, Renata Borovica, Miguel Branco, Stratos Idreos, and Anastasia Ailamaki. 2012. NoDB: efficient query execution on raw data files. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 241–252.
[7] Kaustubh Beedkar, Bertty Contreras-Rojas, Haralampos Gavriilidis, Zoi Kaoudi, Volker Markl, Rodrigo Pardo-Meza, and Jorge-Arnulfo Quiané-Ruiz. 2023. Apache wayang: A unified data analytics framework. *ACM SIGMOD Record* 52, 3 (2023), 30–35.
[8] Yu Cheng and Florin Rusu. 2014. Parallel in-situ data processing with speculative loading. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. 1287–1298.
[9] Benedikt Didrich, Haralampos Gavriilidis, Vasilis Gkolemis, Matthias Boehm, and Volker Markl. 2025. Learning to Accelerate: Tuning Data Transfer Parameters. In *Proceedings of the VLDB 2025 Workshops: AIDB*.
[10] Adam Dziedzic, Aaron J Elmore, and Michael Stonebraker. 2016. Data transformation and migration in polystores. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–6.
[11] Haralampos Gavriilidis, Kaustubh Beedkar, Matthias Boehm, and Volker Markl. 2025. Fast and Scalable Data Transfer Across Data Systems. *Proceedings of the ACM on Management of Data* 3, 3 (2025), 1–28.
[12] Haralampos Gavriilidis, Leonhard Rose, Joel Ziegler, Kaustubh Beedkar, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. 2023. XDB in Action: Decentralized Cross-Database Query Processing for Black-Box DBMSes. *Proc. VLDB Endow.* 16, 12 (2023), 4078–4081.
[13] Haralampos Gavriilidis, Joel Ziegler, Midhun Kaippillil Venugopalan, Benedikt Didrich, Matthias Boehm, and Volker Markl. 2025. Enter the Warp: Fast and Adaptive Data Transfer with XDBC. *Proceedings of the VLDB Endowment* 18, 12 (2025), 5315–5318.
[14] Victor Giannakouris, Nikolaos Papailiou, Dimitrios Tsoumakos, and Nectarios Koziris. 2016. MuSQLE: Distributed SQL query execution over multiple engine environments. In *2016 IEEE International Conference on Big Data (Big Data)*. IEEE, 452–461.
[15] Victor Giannakouris and Immanuel Trummer. 2025. Rethinking Pluggable Federated Query Optimization: From Laptops to Data Warehouses. In *Proceedings of the VLDB 2025 Workshops: Composable Data Management Systems (CDMS)*.
[16] Brandon Haynes, Alvin Cheung, and Magdalena Balazinska. 2016. PipeGen: Data pipe generator for hybrid analytics. In *Proceedings of the Seventh ACM Symposium on Cloud Computing (SoCC)*. 470–483.
[17] Abdulrahman Kaitoua, Tilmann Rabl, Asterios Katsifodimos, and Volker Markl. 2019. Muses: Distributed data migration system for polystores. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1602–1605.
[18] Manos Karpathiotakis, Miguel Branco, Ioannis Alagiannis, and Anastasia Ailamaki. 2014. Adaptive query processing on RAW data. *Proceedings of the VLDB Endowment* 7, 12 (2014), 1119–1130.
[19] Manos Karpathiotakis, Avrilia Floratou, Fatma Özcan, and Anastasia Ailamaki. 2017. No data left behind: real-time insights from a complex data ecosystem. In *Proceedings of the 2017 Symposium on Cloud Computing*. 108–120.
[20] Jiale Lao and Immanuel Trummer. 2026. GenDB: The Next Generation of Query Processing–Synthesized, Not Engineered. *arXiv preprint arXiv:2603.02081* (2026).
[21] Pedro Pedreira, Orri Erling, Maria Basmanova, Kevin Wilfong, Laith Sakka, Krishna Pai, Wei He, and Biswapesh Chattopadhyay. 2022. Velox: Meta's Unified Execution Engine. *Proc. VLDB Endow.* 15, 12 (2022), 3372–3384.
[22] Mark Raasveldt and Hannes Mühleisen. 2017. Don't hold my data hostage: a case for client protocol redesign. *Proceedings of the VLDB Endowment* 10, 10 (2017), 1022–1033.
[23] Mark Raasveldt and Hannes Mühleisen. 2019. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 international conference on management of data*. 1981–1984.
[24] Xiaoying Wang, Weiyuan Wu, Jinze Wu, Yizhou Chen, Nick Zrymiak, Changbo Qu, Lampros Flokas, George Chow, Jiannan Wang, Tianzheng Wang, et al. 2022. ConnectorX: accelerating data loading from databases to dataframes. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2994–3003.
[25] Johannes Wehrstein, Timo Eckmann, Matthias Jasny, and Carsten Binnig. 2026. Bespoke OLAP: Synthesizing Workload-Specific One-size-fits-one Database Engines. *arXiv preprint arXiv:2603.02001* (2026).
[26] Xiangyao Yu, Vijay Gadepally, Stan Zdonik, Tim Kraska, and Michael Stonebraker. 2018. FastDAWG: Improving data migration in the BigDAWG polystore system. In *Heterogeneous Data Management, Polystores, and Analytics for Healthcare (Poly/DMAH @ VLDB)*. Springer, 3–15.
[27] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. 2016. Apache spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65.